Nicholas Chan
nipchan@ucsc.edu
5/02/2021
CSE13s Spring ‘21

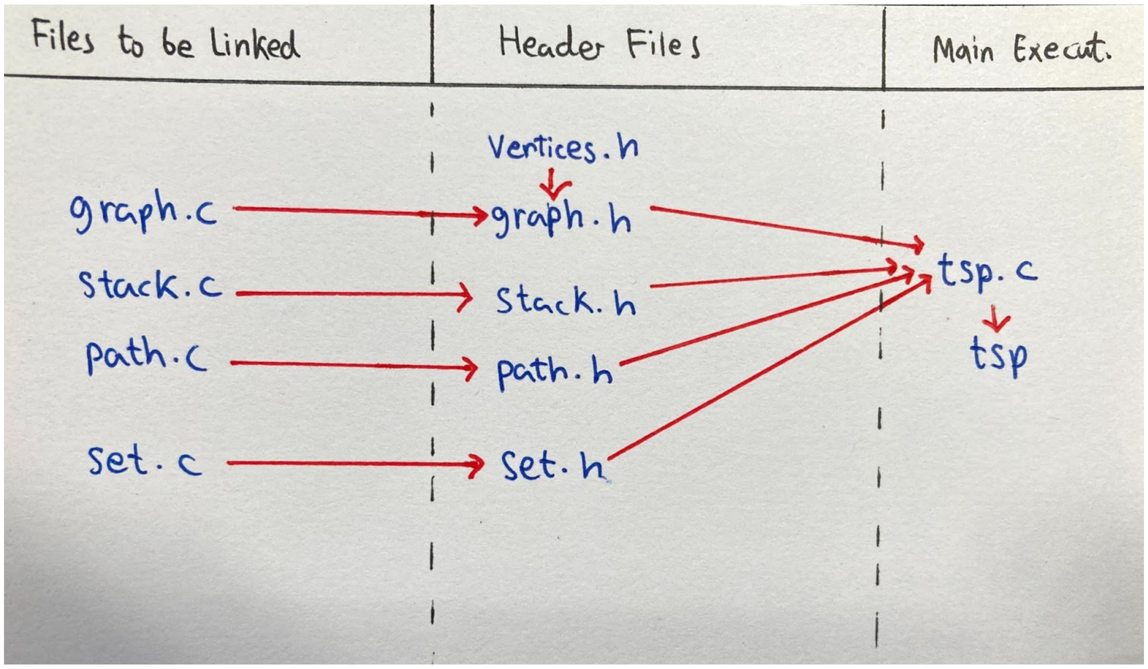
Design Document:
Assignment 4:

Graphs are abstract data types that can be broken down into vertices and edges. The information that graphs convey is similar to the information that waypoints on a map do because of how between two connected waypoints (vertices) there exists a route (edge) with a certain length or magnitude (edge weight). A road trip plotted out on a map that starts and ends in the same place is synonymous to a hamiltonian path in a graph, as long as the condition of never revisiting a once visited waypoint (vertex) is satisfied.

In assignment 4 I have been given the task of implementing graph, path and stack ADTs to be utilized by a recursive function for depth first searching. The depth first search implemented in this assignment will be used to find the shortest hamiltonian path given an input file containing information about a graph, its vertices, its edges and edgeweights. Additionally, this program will have the option to specify whether or not the graph will be undirected or directed. A directed graph would only allow for edges to be traversed in one direction (the way that they are written on the input file) while an undirected graph would allow for edges to be traversed in the direction specified by the input file as well as in the reverse direction (if an edge were written as <1,2,10> and the graph were undirected, it could be traversed from vertex 1 to 2 as well as vertex 2 to 1 with an edge weight of 10. However this is just an example as you cannot revisit an already visited vertex.

Top Level:

How my files will interact



Pseudo Code:

Note: Adapted and inspired from provided pseudo code.

```
Graphs:

Contains
    Integer indicating number of vertices
    Bool specifying if the graph is undirected (true if yes, false if no)
    Array of bools to specify which vertex has been visited (Index=vertex number)
    2D matrix of integers to assign edge weights to edges (idx1=prev vertex,
    idx2=next vertex, value=edge weight)
Call to generate graph
    Allocate memory for graph structure
    Fill vertex array with false bools
    Fill matrix with 0's
Add Edge (stack pointer, integer to insert into stack item array)
    If vertex i and j of edge are within graph bounds
        Assign weight to matrix[i][j]
        return true
    else
        return false
graph has edge
    if matrix[i][j] != 0
        return true
    else
        return false
graph edge weight
    if graph has edge
        return weight at matrix[i][j]
graph mark visited
    if vertex is within graph bounds
        mark visited[vertex] as true
graph mark visited
    if vertex is within graph bounds and is visited
        mark visited[vertex] as false
Destructor(pointer to the graph pointer)
    Free graph pointer
    Set graph pointer to NULL
```

Stacks:

```
Contains
    Integer indicating top position
    Integer indicating capacity
    Pointer to an array of integers

Call to generate stack
Allocate memory for stack structure
    Allocate memory for array of items that will be pointed to by the pointer in each stack

Push (stack pointer, integer to insert into stack item array)
If stack does not have enough memory
    Reallocate original stack structure pointer to a block with twice as much memory
Add item into stack item array

Pop (stack pointer, new variable to assign popped item)
Move top/head pointer from item in array
    Set new variable equal to popped item

Peek  (stack pointer, new variable to assign peeked item)
    Set new variable equal to peeked item (item at items[top-1])

stack copy(destination, source)
    Note: destination is an initialized stack (I'm assuming capacity is same as source's)
    Turns destination's items into a copy of source's items
    Turns destination's top into a copy of source's top

Destructor(pointer to the stack pointer)
    Free items array pointer in the stack structure and set that pointer to null
    Free pointer to stack structure and set that pointer to null
```

Path:

```
Contains
    Stack called vertices which holds visited vertices
    Integer called length which is the sum of all edge weights from visited vertices

Call to generate path
Allocate memory for path structure
    Initializes path stack with capacity of size VERTICES (26)
    Initializes length to a value of 0

path push vertex (Path *p, int v, Graph *g)
    push vertex v onto stack of path p
    increase path length by edge weight of inputted vertex and vertex from peeking stack
        weight will be accessed through g->matrix[peek][push]

path pop vertex (Path *p, int *v, Graph *g)
    pops vertex from stack of path p and gives value to pointer v
    decrease path legnth by edge weight of popped vertex and top vertex after pop
        weight will be accessed through g->matrix[v][peek]

path copy (Path *dst, Path *src)
  Note: destination is a properly initialized path struct\
  copy src's path stack over to dst's path stack
  copy src's path length over to dst's path length
```

Depth First Search:
Note: Eugene's section helped me, I haven’t finished the pseudocode though

```
dfs (Graph g, int v, Path current, Path shortest, char *cities[], FILE *outfile)
    if vertex v is visited
        for all the edges linked to vertex v
            call recursively dfs(g,w) where w is a vertex linked to v through an edge
        for every vertex up till g->vertices
            if graph_has_edge(v,w) and vertex w !visited
                call dfs(g,w) recursively
                pop w off of stack
```

Revised DFS Pseudocode:

```
dfs(g, v=0, curr, shortest, cities)
    markvisited(g,v)
    pathpush(v)
    For possible edge(s) w for v
        If !visited(g,w) and edge exists
            dfs(w)
            markunvisited
        Elif w==0 and edge exists and number of path vertices == number of graph vertices
            dfs(w)
                If curr<shortest
                pathcopy(shortest,curr)
                pathprint(shortest)
    pop(v)
    [END dfs call]
```

Design Process:
Before beginning this lab, I had to familiarize myself with the travelling salesperson problem and refresh myself on recursion as well as graphs. I did this by watching some videos and doing some drawings to visualize the movements of the sorts and then moved onto reading the descriptions of each sort provided in the asgn3 instructions pdf. After this I moved onto setting up the graph, path and stack ADTs, making sure each of them reliably functioned before moving onto the actual depth first search function to be used in solving the tsp problem. I believe that all this preparation was necessary to be able to know how to implement the dfs function in c.

1. Beginning the programming process, I created the graph structure which acted as the setting where the dfs function would act upon
    a. This graph struct took in and stored information for all the vertices as well as all the edges that existed between them
    b. The graph struct also came with the option to mirror itself diagonally if the condition that it was marked as unvisited was specified
2. I then worked on the stack structure
    a. In stack.c I drew some figures of a stack going through pushing and popping to keep track of the movements of the top pointer, size and array items.
        i. After finishing up a draft of stack.c, I ran the print_stack function to test if pushing, popping and the rest of accompanying functions for stack.c worked
            1. Stack worked much like it did in asgn3 with the additions of a new function called stack_peek which was set up almost the same way

            as stack_pop was, with the exception that stack->top was left unchanged in stack_peek
        2. Stack_print was a function based around fprintf and would print output wherever was specified when it would eventually be called in path_print
3. I then worked on the path structure.
    a. This was the last structure that I worked on because of its dependencies on stacks and graphs
    b. The path was essentially a stack with the extra parameter of a stored path length
    c. The path struct also had a print_path function tied to it which would call on stack_print to print out the shortest known (at least at the tim of execution) hamiltonian paths in a graph, given a valid set of vertices
4. I then worked on setting up the recursive dfs function
    a. I first drew the function out on paper and made drawings to fully understand the actions necessary for dfs.
    b. The logic for dfs was something I spent quite a bit on
    c. Some important things to consider which I learned from finishing dsf were to make a call to free for every malloc/calloc and to match every push call in a dfs call with a pop call.
    d. I also realized that a way to shorten the needed recursive calls for a function was to end the dfs function immediately if a path length was already > that the shortest path length recorded.
5. Finally I created a Makefile to properly compile the source code of my test harness and source files.
    a. This called for -lm to link my files for compilation

I thought this assignment was pretty challenging especially with the need for memory management and recursion, but finishing was satisfying. I also really liked working with graphs on this assignment. The freedom we were allowed for modifying the source and some header files was pretty fun at times as I could freely modify some of the given prototyped functions in the asgn4 pdf. If there were something I would do differently the next time around for this assignment, I would set up a path and stack print function specifically for debugging as the path and stack print functions used in this assignment were actually used to write output after finishing a depth first search on a graph.

References:

- https://www.youtube.com/watch?v=c8P9kB1eun4
- https://en.wikipedia.org/wiki/Graph_(abstract_data_type)#:~:text=In%20a%20computer%20science%2C%20a,graph%20theory%20within%20a%20math.&text=These%20pairs%20are%20known%20as,are%20also%20known%20as%20arrows.